ソフトウエア開発における効果的なソースコード解析手法*

酒井 正志† 河崎 文雄† 板東 由美† 渡部 淳一† 宮崎 肇之†
(株)日立製作所 情報・通信グループ プロジェクトマネジメント統括推進本部‡

1. はじめに

社会インフラを支える業務システムでは、高信頼なソフトウェア開発が必要である。しかし、当事業部では、プログラミングレベルの不具合が原因で後工程や本番で発見される事例が増加している。そのため、単体テスト工程での品質確保が現状の重要な課題である。その解決案の一つとして、ソースコードの解析を効果的に実施することで品質面、管理面の効果向上を図ることができると考える。そこで、弊社では、高信頼性なソフトウェア開発の実現を目的として、新たにソースコード解析手法を検討した。本稿では、検討したソースコード解析手法について紹介する。

2. ソースコード解析

2.1 ソースコード解析の概要と弊社の取り組み

ソースコードに対する解析は、動作不良などを引き起こす可能性を減らす目的で実施される。また、解析には、品質上問題となるパターンなどを定義したコーディングルールが用いられる。そして、コーディングルールに違反している箇所を発見した場合、開発者は該当箇所を修正する。ソースコード解析作業と修正作業を繰り返すことで、ソースコードの品質向上を図ることができる。

弊社では、ソースコード解析作業を、誰もが実施する開発プロセスとして、プログラミング工程の手順に取り入れている。また、解析にはソースコード解析ツールを適用している。ソースコード解析ツールとは、ソースコードの静的解析を機械的に実施するツールである。本ツールの適用により、開発者のスキル・裁量に依存していた作業の信頼性向上を図ることが可能となる。

2.2 静的解析ツールの問題点

2.1節で述べたソースコード解析ツールは、以下に示す問題点を抱えていた。

(1) 静的解析の結果を、編集中のソースコードに対して即座に反映できない。
(2) ソースコードの規模に応じて、解析ツールによる指摘件数が膨大な量になる。このため、開発者がどの警告から修正すべきか判断ができない。

なお、Java、COBOL、.NET といった開発言語で作成されたソースコードを機械的に解析する製品は幾つか知られている。しかしながら、我々の限られた調査の範囲では、上記の問題を解決できる製品は見当たらなかった。

そこで、上記の問題を解決するためのソースコード解析手法を検討することとした。本稿では特に、上記の 2.2 項(1)に対して検討した内容を以下に述べる。

3. ソースコード解析方式の検討

3.1 従来のソースコード解析における問題

従来のソースコード解析における問題について、図 1、図 2を用いて説明する。
図 1は、2 人の開発者がそれぞれ担当のプログラムを 1 本ずつ担当していることを示す。図 2は、開発するプログラムの構成図を示す。プログラムＢは、プログラムＡのサブモジュールであることを示す。

従来、複数のプログラムに跨る静的解析処理を実施する場合、必要なプログラムを全て取得した上で実施する必要があった。例えば、[開発者Ｂ]が担当する[プログラムＢ]の解析を行う場合は、[プログラムＡ]も取得する必要があった。その結果、[開発者Ｂ]が解析処理に対する作業負荷がかかってしまう。加えて、全てのソースコードに対して解析処理を実施するため、処理時間も増大する。

また、[開発者Ａ]の進捗によっては、[開発者Ｂ]に待ちの時間が発生する。結果、作業遅延の原因ともなりうる。

---

* Effective source code analysis technique in software development
† Sakai Masashi, Kawasaki Fumio, Bandou Yumi, Watanabe Junichi, Miyazaki Tadashi
‡ Project Management & Software Engineering PromotionDepartment Project & Process Management Division

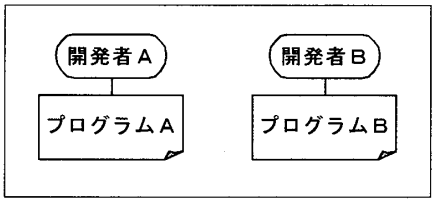

図 1　開発者と開発プログラムの関係

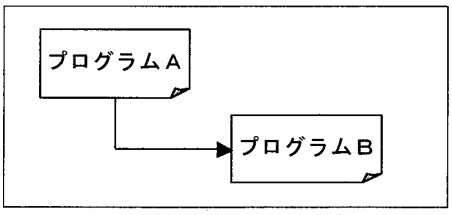

図 2　プログラムの構成図

3.2　対策

3.1 節に挙げた問題を解決するために検討したソースコード解析手法の流れおよび対策内容について以下に述べる。

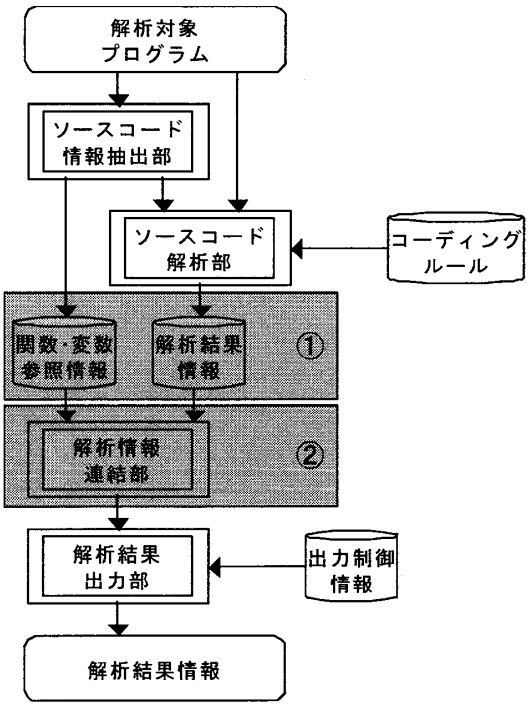

図 3　ソースコード解析処理の流れ

まず、ソースコード解析に必要な情報を蓄積する記憶手段を持たせることとする（図 3中の①）。なお、必要な情報とは、解析結果情報やソースコードに対する関数・変数の参照情報などを指す。解析処理の度に、解析結果情報、ソースコード関連情報を蓄積していく。例えば、[プログラムB]の解析を先に実施した場合とする。解析処理終了後、[プログラムB]の情報が記憶手段に蓄積される。続いて、[プログラムA]の解析を実施する。従来は、ここで[プログラムB]を取得し、全てのプログラムに対して解析処理を実施していた。しかし、記憶手段を設けることで、蓄積された情報をもとに解析処理を実施できる。

これにより、ソースを入手する手間を省くことができる。すなわち、開発者の作業負荷軽減を狙うことが可能になると期待できる。また、全てのソースコードを読み込む必要も無いため、処理時間の短縮にも期待ができる。

また、入力されたソースコードの解析結果によって、蓄積されている解析結果情報が影響を受けるかどうかを判断する解析結果連結手段（図 3中の②）を設ける。プログラム単体で解析した場合と、関連プログラムの情報も参照した上での解析結果が異なるかどうか判断する。例えば、後から解析した[プログラムA]の解析結果が、既に蓄積されている[プログラムB]の解析結果情報に影響するかを判断する。[プログラムA]の解析結果によって、本来指摘されなくても良い警告が存在する場合は、蓄積情報から削除する。

これにより、静的解析結果情報を編集中のソースコードに対しても反映することが可能となる。すなわち、不良の早期発見につながると期待できる。また、常に精査された情報で解析処理が可能となる。これにより、静的解析結果が膨大に出力されることも抑えることが可能であると考える。

4.　結論

ソフトウェア検証時におけるソースコード解析手法について検討し、紹介した。これにより、ソフトウェア開発における品質面、管理面での効果向上が期待できると考える。

5.　今後の課題

現在のところ、本提案を適用した事例は存在しない。しかし、Java、COBOL などによる開発プロジェクトは存在するため、今後は、本提案を適用した場合の効果向上を評価すると共に、手法の改善を図っていく。